COMPILER CONSTRUCTION FOR DIGITAL COMPUTERS

David Gries

COMPILER CONSTRUCTION FOR DIGITAL COMPUTERS

V

David Gries
Cornell University

John Wiley & Sons, Inc.
New York • London • Sydney • Toronto

Library of Congress Catalogue Card Number: 74-152496

ISBN 0-471-32776-X

Printed in the United States of America.

10 9 8 7 6 5 4 3 2

Preface

Compilers and interpreters are a necessary part of any computer system -- without them, we would all be programming in assembly language or even machine language! This has made compiler construction an important, practical area of research in computer science. The object of this book is to present in a coherent fashion the major techniques used in compiler writing, in order to make it easier for the novice to enter the field and for the expert to reference the literature.

This book is oriented towards so-called syntax-directed methods of compiling. In fact, over one third of the book is devoted to the subject of formal language theory and automatic syntax recognition. I feel very strongly that anybody involved in compiler writing should have a basic knowledge of the subject. This does not mean that every compiler should be written using automatic syntax methods. There are many programming languages where these methods are not suitable. But a basic knowledge of formal language theory will give the compiler writer more insight into what is happening inside his compiler, and should help him design and program more systematically and efficiently.

Syntax analysis, however, is only a small part of compiler construction, and I have included chapters on all the major topics -- symbol table organization, error recovery, code generation, code optimization, and so forth. Several topics (e.g. conversion of constants, incremental compilers) have been omitted in order to keep the size of the book reasonable.

The book is meant to serve two needs; it can be used as a self-study and reference book for the professional programmer interested in or involved in compiler construction, and as a text in a one-semester course in compiler writing. In fact, the book covers all the topics (and more) listed for the course I5 (Compiler Construction) recommended by the ACM curriculum committee in the March 1968 issue of the Communications of the ACM.

The reader should have at least one year of experience programming in a high-level language (eg. FORTRAN, ALGOL, PL/I), and in an assembly language, and should be able to read and understand ALGOL programs. In some parts, elementary Boolean matrix theory is assumed (with a short introduction).

It is assumed the reader knows what a set is, what the union of two sets is, and so on. Beyond this, the reader should have the mathematical experience of, say, a sophomore or junior math major.

The need for experience with a high-level language is obvious; the book is about translating programs written in such languages. Experience with an assembly language is similarly necessary. Assembly language experience is more important, however, for the maturity and understanding of how computers work that it provides. Actually, we will have little to do with any specific assembly language. The few IBM 360 assembly language programs scattered throughout the book can be skipped over without loss of understanding.

A compiler is just a program written in some language. Hence, examples of parts of compilers must be given in some programming language, and I have chosen an ALGOL-like language for its readability. The examples are usually very short, so that they can be followed easily. Where too much detail will cause us to lose sight of the problem at hand, I have taken the liberty to write English instead of ALGOL. I have checked these program segments quite carefully by hand, but the reader is warned that they have not all been debugged on a computer!

A brief description of the bastard ALGOL used appears in the appendix. This description is short and relies heavily on a knowledge of ALGOL. Should the reader not be familiar with ALGOL and syntax descriptions of languages, it is suggested that he wait until after studying chapter 2 to read the appendix.

There is more material than can usually be covered in a one-semester course. The following minimum set is suggested:

Chapter 1.	Introduction
Chapter 2.	Grammars and languages; omit section 7
Chapter 3.	Scanners; omit sections 4,5,6
Chapter 4.	Top-down parsing; especially section 3
Chapter 5.	Simple precedence grammars
Chapter 8.	Runtime storage organization; omit sections 6 and 9
Chapter 9.	Organizing symbol tables
Chapter 10.	The data in the symbol table; omit section 2
Chapter 11.	Internal forms of the source program; omit sections 4,5
Chapter 12.	Introduction to semantic routines
Chapter 13.	Semantic routines for ALGOL-like constructs; omit section 6
Chapter 14.	Allocation of storage to runtime variables; omit section 3
Chapter 16.	Interpreters
Chapter 22.	Hints to the compiler writer

You will notice that I emphasize the simple precedence technique for syntax analysis. This is not because it is the best (it is probably the worst), but it is the easiest to teach. Should more time be available, the instructor is encouraged to add his favorite bottom-up syntax method: operator precedence (section 6.1), higher order precedence (section 6.2), transition matrices (section 6.4), production language (chapter 7), or any other not covered.

The order of presentation may also be changed. In fact, when teaching a course, it is best to break up the study of syntax theory with some practical material. Chapter 8 on runtime storage administration is independent, while chapters 9, 10, 11, and 16 on symbol tables, internal source program forms, and interpreters can be studied in that order at any time.

Chapter 21 deserves special mention. It is a collection of assorted facts and opinions that a compiler writer should be familiar with. They don't belong anywhere else, or are two important to be buried in some other chapter. The reader should browse through this chapter from time to time and read the sections of current interest.

A compiler-writing course should be a laboratory course. Students should write and debug a compiler or interpreter for some simple language, in groups of one to three people. Only then will they really understand what goes into a compiler. An interpreter is best, since the students don't have to worry about messy machine language details; the ideas are important, not the details. Following this line, the whole project should be programmed in a high-level language. My experience is that PL/I or an ALGOL-like language is better than FORTRAN. Compilers in FORTRAN tend to be larger and much more difficult to read. A translator writing system should be used if available.

To produce some variability and creativity, start with a basic, simple language containing integer variables, assignment statements, expressions, labels and branches, conditional statements, and finally simple read and write statements. Then let each group extend it by adding one or two features. Examples are arrays, records (structures), different data types, block structure, procedures, macros, and iterative statements.

The compiler can be written and checked out in stages as the course progresses. First the scanner, then the syntax analyzer, then the symbol table routines, and finally the semantic routines. The interpreter itself can be designed and implemented as soon as the chapters dealing with it have been covered. In this way, the work is spread out evenly throughout the semester, and is not bunched up at the end.

Most references to publications are given in a section at the end of each chapter, although some occur within other sections. The appearance of a name of a person is automatically a reference to a publication listed in the bibliography. This bibliography is in alphabetical order by author, and within each author it is arranged chronologically. If a person has more than one publication the reference will appear in the form <name>(<year>), where the year refers to the year of publication. An example is Gries(68). Should an author have more than one publication in one year, the first listed in that year is referenced by (<year>a), the second by (<year>b), and so on. Thus Floyd(64b) refers to Floyd's paper on the syntax of programming languages.

Except for headings and some figures, these notes were produced on the IBM 360/65, using a program called FORMAT written by Gerald M. Berns(69). The author is also indebted to John Ehrman, who made several important changes and additions to the program. The use of FORMAT made it easier to edit the original material and distribute it to students at various stages. However, it forced me to deviate somewhat from conventional notation. The two main changes are the following. The printer chain which printed the book has no subscripts and no superscripts (except for ^{0} through ^{9}). Exponentiation is therefore written using the operator !. That is, c!b means c to the bth power. The lack of subscripts forced me to write a sequence of n symbols as S[1], S[2], ..., S[n]. Where its meaning is obvious, we write this simply as S1, S2, ..., Sn.

Sections of this book originated as lecture notes in compiler writing courses at Stanford and Cornell, and I have had the opportunity to use the notes in revised form in short courses at the Michigan Summer School at Ann Arbor, Cornell, and the 1970 International Seminar in Advanced Programming Systems in Israel. I am indebted to the students in these courses for their critical comments on these notes. I have had helpful advice from a number of people; among them are Richard Conway, Jerry Feldman, John Reynolds, Bob Rosin, and Alan Shaw. My sincere appreciation goes to Steve Brown, who read the manuscript carefully and thoroughly, found many mistakes, and gave valuable comments and criticisms. Finally, I would like to thank my wife, who showed amazing patience and understanding while I was writing this book.

Table of Contents

Chapter 1.
Introduction

1.1 COMPILERS, ASSEMBLERS, INTERPRETERS

A _translator_ is a program which translates a _source program_ into an equivalent _object program_. The source program is written in a _source language_, the object program is a member of the _object language_. The execution of the translator itself occurs at _translation time_.

If the source language is a high-level language like FORTRAN, ALGOL, or COBOL, and if the object language is the assembly language or machine language of some computer, the translator is called a _compiler_. Machine language is sometimes called _code_; hence the object program is sometimes called the _object code_. The translation of the source program into the object program occurs at _compile-time_; the actual execution of the object program at _runtime_.

An _assembler_ is a program which translates a source program written in assembly language into the machine language of a computer. Assembly language is quite close to machine language; indeed, most assembly language statements are just symbolic representations of machine language statements. Moreover, assembler statements usually have a fixed format, which makes it easier to analyze them. There are usually no nested statements, blocks, and so forth.

An _interpreter_ for a source language accepts a source program written in that language as input and executes it. The difference between a compiler and an interpreter is that the interpreter does not produce an object program to be executed; it executes the source program itself.

A pure interpreter will analyze a source program statement _each_ time it is to be executed in order to discover how to perform the execution. This of course is very inefficient and is not used very often. The usual method is to program the interpreter in two phases. The first analyzes the complete source program, much the way a compiler does, and translates it into an internal form. The second phase then interprets or executes this internal form of the source program. The internal form is designed to minimize the time needed to "decode" or analyze each statement in order to execute it.

As explained earlier, a compiler is itself just a program written in some language -- its input is a source program and its output is an equivalent object program. Historically, compilers were written in the assembly language of the computer at hand. In many cases this was the only language available! The trend is, however, to write compilers in high-level languages, because of the reduced amount of programming time and debugging time, and the readability of the compiler when finished. We also find many languages designed expressly for compiler writing. These so-called "compiler-compilers" are a subset of the "translator writing systems" (TWS); we discuss these briefly in chapter 20.

This book serves to introduce you to compiler construction. The problems of interpreters will also be discussed; this will add comparatively little to the book since most techniques used in compiler construction are also used in writing interpreters. We will not discuss assemblers, but anyone who understands compiler construction should have no trouble understanding what an assembler does and how it performs its job.

You will not find a complete compiler anywhere in this book. The idea is not to see how I write one particular compiler, but to learn how to write your own. You will of course find examples and discussions of many (but of course not all - I do not even presume to say most) of the techniques and methods used in compiler construction. Examples will be programmed in a bastard ALGOL language, a brief description of which appears in the appendix. If you are using this book as a text in a course, hopefully you will write your own compiler or interpreter in ALGOL, FORTRAN, PL/1 or other high-level language; this is the best way to learn about compiler construction.

1.2 A BRIEF LOOK AT THE COMPILATION PROCESS

A compiler must perform an analysis of the source program and then a synthesis of the object program. First decompose the source program into its basic parts; then build equivalent object program parts from them. In order to do this, the compiler builds several tables during the analysis phase which are used during both analysis and synthesis. Figure 1.1 shows the whole process in more detail; dotted arrows represent flow of information, while solid arrows indicate program flow. Let us briefly describe the different parts of a compiler.

Tables of Information

As a program is analyzed, information is obtained from declarations, procedure headings, for-loops, and so forth, and saved for later use. This information is detected at a local level and collected so that we have access to it from all parts of the compiler. For example, it is necessary to know with each use of an identifier how that identifier was declared and used elsewhere. Exactly what must be saved depends of course upon the source language, the object language, and how sophisticated the compiler is. But every compiler uses a symbol table (sometimes called an identifier list or name table) in one form or another. This is a table of the identifiers used in the source program, together with their attributes. The attributes are the type of the identifier, its object program address, and any other information about it which is needed to generate code.

What other information must be collected? We will most likely need a table of constants used in the source program. This table will include the constant itself and the object program address assigned to it. We may also need a table of for-loops showing the nesting structure and the loop variables, information about FORTRAN-like EQUIVALENCE statements, and a list of the object program sizes of each procedure being compiled. When designing a compiler, one cannot determine the form and content of the information to be collected until the object code for each source program statement and the synthesis part of the compiler have been thought out in some detail. Much depends on how much code optimization is going to be performed.

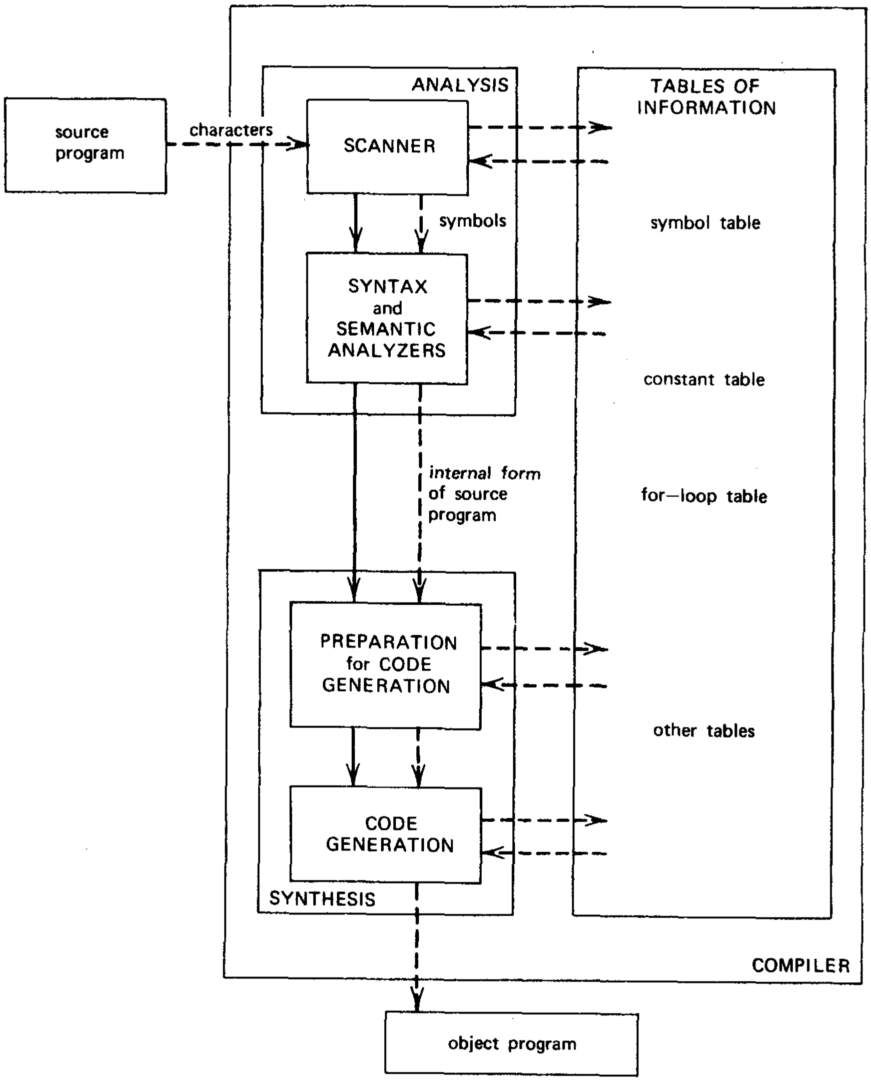

FIGURE 1.1. Logical Parts of a Compiler.

The Scanner

The scanner -- the simplest part of the compiler -- is sometimes called a <u>lexical analyzer</u>. It scans the characters of the source program from left to right and builds the actual <u>symbols</u> of the program -- integers, identifiers, reserved words, two-character symbols like ** and //, and so forth. (In the literature, the terms <u>token</u> and <u>atom</u> are sometimes used for <u>symbol</u>.) These symbols are then passed on to the actual analyzer. Comments can be deleted. The scanner may also put the identifiers into the symbol table and perform other simple tasks that can be done without really analyzing the source program. It can do most of the macro processing for macros which allow only a textual substitution.

The symbols are usually passed by the scanner to the analyzer itself in an internal form. Each delimiter (reserved word, operator or punctuation mark) will be represented by an integer. An identifier or constant can be represented by a pair of numbers. The first, different from any integer representing a delimiter, will indicate "identifier" or "constant"; the second will give the address or index of the identifier or constant in some table. This allows the rest of the compiler to operate in an efficient manner with fixed-length symbols rather than variable length strings of characters.

The Syntax and Semantic Analyzers

These analyzers do the actual hard work of disassembling the source program into its constituent parts, building the internal form of the program and putting information into the symbol table and other tables. A complete syntax and semantic check of the program is also performed.

The standard analyzers are controlled by the syntax of the program. In fact the trend has been to separate the syntax from semantics as much as possible. When the syntax analyzer (parser) recognizes a source language construct it calls a so-called <u>semantic procedure</u> or <u>semantic routine</u> which takes the construct, checks it for semantic correctness, and stores necessary information about it into the symbol table or the internal form of the program. For example, when a simple declaration is recognized, a semantic routine will check the declared identifiers to make sure they have not been declared twice and will add them to the symbol table with the declared attributes. When an assignment statement of the form

<variable> := <expression>

is recognized, a semantic routine will check the <variable> and <expression> for type compatibility and will then put the assignment statement into the internal program.

The Internal Source Program

The internal representation of the source program depends largely on how it is to be manipulated later. It may be a tree representing the syntax of the source program. It may be the source program in something called Polish notation. Another form used is a list of (operator, operand, operand, result) quadruples, in the order in which they are to be executed. For example, the assignment statement "A = B + C * D" would appear as

```
*, C, D, T1
+, B, T1,T2
=, T2, A,
```

where T1 and T2 are temporary variables created by the compiler. The operands in the above example would not be the symbolic names themselves, but pointers to (cr indexes to) the symbol table elements which describe the operands.

Preparation for Code Generation

Before code can be generated, it is generally necessary to manipulate and change the internal program in some way. Runtime storage must be allocated to variables. In FORTRAN, COMMON and EQUIVALENCE statements must be processed. One important point included here is the <u>optimization</u> of the program in order to reduce the execution time of the object program.

Code Generation

This is the actual translation of the internal source program into assembly language or machine language. This is perhaps the messiest and most detailed part, but the easiest to understand. Assuming we have an internal form of quadruples as outlined above, we generate code for each quadruple in order. For the three quadruples listed above we could generate, on the IBM 360, the assembly language

```
    L       5,C     Put C in register 5
    M       4,D     The result of the mult. is in regs. 4,5
    A       5,B     Now add B to the result of the mult.
    ST      5,A     Store the result
```

In an interpreter, this part of the compiler would be replaced by the program which actually executes (or interprets) the internal source program. The internal form for the source program in this case would not be too much different.

Figure 1.1 represents a logical connection of the compiler parts rather than a time connection. All four of the logically successive processes -- scanning, analysis, preparation for code generation and code generation -- can be performed in the order given by Figure 1.1, or they could be performed in a parallel, interlocked manner. One criterion for this is the amount of available memory. It is often advantageous or even necessary to have several passes (core loads). Hopefully in these cases the "other information" can be kept in memory to save I/O time. Other criteria are the goals of the project. How fast should the compiler itself be? How fast should the object program be? How much debugging facilities should the object program provide? Another factor is the number of people on the project. The more people, the more passes there are likely to be, so that each can be responsible for a distinct and separate part.

It is also true that not all the parts need be used. In a one-pass compiler, the internal form of the program is not necessary, while the preparation and code generation parts are fused with the semantic routines of the semantic analyzer. A typical one-pass scheme is given in Figure 1.2. The syntax analyzer calls the scanner when it needs a new symbol and calls a procedure when a construct is recognized. This procedure does semantic checking, storage allocation, and code generation for the construct before returning to the parser.

Not all languages are structured so that they can be translated by a one-pass compiler.

One may well ask where the main difficulties lie in implementing a compiler. The scanner is almost trivial and is well understood. Syntax analyzers are also fairly well understood for the simple formal languages we deal with. In fact, this part can be largely automated. (Since syntax has been formalized, much of the research in compiler writing has dealt with it instead of semantics.) The hardest and "dirtiest" parts are semantic analysis, program preparation and code generation. All three are interdependent and must be designed together to a large extent, and the design can change radically from one object program language and machine to another.

With this brief introduction we are ready to begin with our first subject - formal language theory and its application to compiler construction. If you wish (it is not necessary), glance over the next section which gives some examples of existing compilers in order to reinforce the material presented here.